NRL DOCUMENT: 0920-142 (Preliminary)

1

AD-A236 286

INTERACTIVE GRO/OSSE REDUCTION ENVIRONMENT (IGORE) DESIGN DESCRIPTION

GAMMA RAY OBSERVATORY

ORIENTED SCINTILLATION SPECTROMETER EXPERIMENT

Northwestern University
Evanston, IL

N00173-85-C-2501

Critical Design Review Version

Release Date: February 3, 1989

Author: David Grabelsky

DTIC
ELECTE
JUN 07 1991
S B D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

91 6 6 068

91-01445

DTIC COPY INSPECTED 6
ssion For
GRA&I
TAB
nounced
ification
per letter
ribution/
ilability Codes
Avail and/or
Dist Special
A-1

1 INTRODUCTION

1.1 Purpose

This document describes the detailed software design of the Interactive GRO/OSSE Reduction Evironment (IGORE). IGORE is an interactive scientific data analysis program for maniputlation, reduction, and analysis of OSSE science data. The software requirements of IGORE are described in document [1] (see list below); these requirements delineate the capabilities and functions which IGORE shall provide each user in order to carry out the data analysis tasks. The software design used to implement the requirements is described in this document.

1.2 Scope

The design description contained in this document pertains mainly to utilities and functions of general use in IGORE, such as passing data between applications, declaration of structures and records, etc. Specific scientific data analysis applications, such as spectral summation, resampling of spectral channels, and deconvolution are beyond the scope of this document.

1.3 Applicable Documents

The documents listed below will be referred to as needed in this document; they will be referenced according to their number in the list.

1. IGORE Software Requirements Specifications; Gamma Ray Observatory Oriented Scintillation Spectrometer Experiment; NRL Document 0926-129. Author: David Grabelsky.

2. IDL User's Guide. Research Systems Inc.

3. Data Analysis System Requirements Specification; Gamma Ray Observatory Oriented Scintillation Spectrometer Experiment. Author: Mark S. Strickman.

4. Preliminary Data Analysis Plan; Gamma Ray Observatory Oriented Scintillation Spectrometer Experiment. Author: Mark S. Strickman.

5. Spectral Data Base (SDB) User's Guide; OSSE Software Library; Gamma Ray Observatory Oriented Scintillation Spectrometer Experiment. Authors: Rod S. Hicks, Nina M. Sweeney, and Jack D. Daily.

6. Fit Data Base (FDB) User's Guide; OSSE Software Library; Gamma Ray Observatory Oriented Scintillation Spectrometer Experiment. Author: David Kuo.

1.4 List Of Acronyms

AFE	Applications Frontend
AOE	Abort On Error
COW	Continue On Warning
DBMS	Data Base Management System
EMS	Eat My Shirt
FDB	Fit Data Base
GRO	Gamma Ray Observatory
HLCL	High Level Command Language
ICH	IGORE Condition Handler
IGORE	Interactive GRO/OSSE Reduction Environment
IPOA	Indiscriminate Proliferation Of Acronyms
LOA	List Of Acronyms
NRL	Naval Research Lab
NU	Northwerstern University
OSSE	Oriented Scintillation Spectrometer Experiment
PIF	Program Interface
SDB	Spectral Data Base
UIF	User Interface

2 DESIGN OVERVIEW

2.1 High Level Command Language: IDL

2.2 Structures And Records

2.3 Application Frontend (AFE)

2.4 General Tables Facility

2.5 Saving And Restoring Environment

2.6 Journaling

2.7 IGORE Condition Handler

2.8 Preprocessors

3 DEFINITIONS AND TERMINOLOGY

3.1 Descriptors In IGORE

Many of IGORE's functions are implemented using structures of various types to store and relay information about the data being processed. These structures shall be referred to as descriptors since they contain information similar to that contained in standard VAX descriptors. Six basic descriptor types are used; each is described in the following subsections.

3.1.1 VAX Descriptor: VAX_DESCR

This shall refer to the standard VMS descriptor used for VAX data types. Only the prototype portion (first two longwords) is used. The following structure can be used to access this descriptor:

```
structure /vax_descr/
   integer*2     length          !bytes per data element
   byte          type            !VAX data type
   byte          class           !scalar, array, etc.
   integer*4     pointer         !pointer to start of data
end structure
```

For arrays, offsetting one longword beyond POINTER will access ARRSIZ, the total number of bytes in the array.

3.1.2 IDL Descriptor: IDL_DESCR

This shall refer to the standard descriptor used for all IDL variables (see section 4 below). The following structure can be used to access this descriptor:

```
structure /idl_descr/
   byte          type            !IDL data type
   byte          flags           !constant, array, temporary, etc.
   integer*2     link            !for IDL system use only
   integer*4     valu_word1      !depends on type, flags
   integer*4     valu_word2      !depends on type, flags
   byte          nchar           !no. characters in name
   character*15 name             !variable name = name(1:nchar)
end structure
```

For scalars (single 1, 2, 4, or 8 byte data elements), the value is embedded in the descriptor starting at the first byte of VALU_WORD1. For arrays of all types, VALU_WORD1 is a pointer to a standard VAX array descriptor. For scalar strings, a standard VAX dynamic string descriptor is embedded in the two value words, starting at the first byte of VALU_WORD1. IDL only allocates up to NCHAR bytes (the actual length of the variable's name, .le.15) beyond NCHAR for the name. Attempting to access bytes beyond NCHAR will produce unpredictable results.

3.1.3 Data Descriptor: DATA_DESCR

This descriptor shall be used to manage standard OSSE data in a very compact form. The information carried in this descriptor shall be the address and size (in bytes) of the associated data if the data are resident in memory, a pointer to an entry in the FILE_TAGWORD TABLE if the data reside on disk, the data type (SDB, FDB, etc.), and the number (up to eight) and size of the dimensions of the data. The following structure can be used to access DATA_DESCRs:

```
structure /data_descr/
   integer*4     pointer        !address of memory-resident data
   integer*4     nbytes         !total memory allocation for data
   character*16  data_type      !standard OSSE data type
   byte          tbl_index      !entry no. in file_tagword table
   byte          reserved_1     !reserved for future use
   byte          reserved_2     !reserved for future use
   byte          reserved_3     !reserved for future use
   integer*4     dim_1          !first dimension or 0 if scalar
   integer*4     dim_2          !second dimension or 0 if beyond last
       .             .                     .
       .             .                     .
       .             .                     .
   integer*4     dim_8          !eigth dimension or 0 if beyond last
end structure
```

When OSSE data are maniputlated as header+data (as opposed to simply header or simply data), the actual data will be represented as DATA_DESCR embedded in the header. Routines for cracking data DATA_DESCRs shall be provided.

3.1.4 IGORE Structure Descriptor: STRUC_DESCR

This shall refer to descriptors of native IGORE structures. The purpose of STRUC_DESCRs is to enable manipulation (creation, cracking, etc.) of all records defined to be a type associated with a given STRUC_DESCR. The basic structure of each such descriptor will be a character string consisting of delimited subfields containing the ASCII name of each field of the associated structure and a descriptor of the data associated with that subfield. STRUC_DESCRs may be of variable length, depending on the number and type of subfields in the associated structure. Once a particular STRUC_DESCR is "loaded" into IGORE, any number of records of the type described by that STRUC_DESCR may be interactively declared.

STRUC_DESCRs will be created by a preprocessor that parses source files with familiar FORTRAN definitions made with the STRUCTURE statement. A system library of STRUC_DESCRs will be maintained in a readonly shared global common; separate user libraries may also be maintained. First use of a given STRUC_DESCR will cause it to be loaded into memory. When a user references a STRUC_DESCR type which is not already loaded, IGORE will always search the user's library first for that type. This will allow users to change the default definition of system stucture types (identified by mnemonic).

3.1.5 Structure Subfield Descriptors: REC_FLD_DESCR

This shall refer to the descriptor of a given subfield of an IGORE structure. Strings of delimited REC_FLD_DESCRs for a given structure make up the STRUC_DESCR for that structure. REC_FLD_DESCRs have variable lengths, depending on the name and type of the associated subfield.

The REC_FLD_DESCR prototype consists of: i) a "^" delimiter; ii) the ASCII name of the field (up to 15 characters); iii) a "%" delimiter; iv) a fixed length descriptor portion; v) the number dimensions (up to 8) if the associated varaible is an array; and vi) the sizes of each of the dimensions. Access to the descriptor of a given subfield uses the FORTRAN character INDEX function to find the delimited name of the field; the maximum-lengthed field (i.e., assuming all 8 dimensions are used) following the name is then accessed. This descriptor portion is then cracked in order to get to the actual data.

Access to the descriptor portion following the delimited name can be made with the following structure:

```
        structure /rec_fld_descr/
           byte          type           !data type
           integer*2     size           !bytes per element
           integer*4     offset         !byte offset from beginning of record
           byte          address_mode   !direct or indirect
           byte          ndims          !number of dimensions (1 - 8)
           integer*4     d1             !size of dimension #1
               .          .                   .
               .          .                   .
               .          .                   .
           integer*4     d8             !size of dimension #8
        end structure
```

Accessing more than NDIMS dimensions will yield unpredictable results.

Three addressing modes shall be distinguished: i) Direct (ADDRESS_MODE = 0); ii) Indirect/pointer (ADDRESS_MODE = 1); and iii) Indirect/descriptor (ADDRESS_MODE = 2). In Direct mode access, a data item of TYPE, SIZE, NDIMS, and D1,...Dn (n.le.8) is located at OFFSET bytes in contiguous memory from the starting byte of the actual record. In Indirect/pointer mode access, the data item that is stored contiguosly with the rest of the record is a longword pointer to the actual location of the data associated with the field. In this case, TYPE, SIZE, NDIMS, and D1,...Dn (n.le.8) refers to the actual data associated with the field. In Indirect/descriptor access mode, the data item that is stored contiguosly with the rest of the record is a DATA_DESCR of data associated with the field.

Default access to all fields is to the actual data associated with the field. That is, if data are stored in either of the two indirect modes, the actual address of the associated data is resolved and those data are accessed. Access to the actual pointer value or DATA_DESCR can be made by appending a "@" character to the name of the field (a warning will be issued in this case if the field contains actual data, i.e. is flagged ADDRESS_MODE = direct).

3.1.6 Dynamic IGORE Record Descriptor: DYNAM_REC_DESCR

This shall refer to a record containing information needed to access a given declared IGORE record variable. Every record declared dynamically in IGORE will have a DYNAM_REC_DESCR associated with it. IGORE will maintain a these records in the DYNAM_REC_DESCR_TABLE, adding new entries each time a new record is declared. The DYNAM_REC_DESCR can be accessed with the following structure:

```
structure /dynam_rec_descr/
   integer*2     table_index     !address in the table for access
   byte          nchar           !no. of character in name
   character*15  name            !record name = name(1:nchar)
   byte          rec_type        !record type (translated mnemonic)
   integer*4     rec_ptr         !pointer = address of first byte
   integer*4     total_size      !total number bytes (.ge. 1 record)
   integer*4     rec_size        !no. bytes per record
   integer*2     nrec            !no. of records (1-dim arrays only)
   integer*4     str_descr_ptr   !pntr to STRUC_DESCR for this rec_type
end structure
```

All IGORE records will be represented in IDL as IDL scalar or vector (1-dimensional arrays) longword variables. Each longword value will consist of an I*2 number pointing to the table index of the associated record (TABLE_INDEX) and an I*2 number corresponding to the record-vector index. For example, a single record element variable will be an IDL scalar longword; the first I*2 number will point to the table index for this variable, the second will be set to one (1). An array of records (1-dimensional) will be an IDL longword array. The first I*2 portion of each array element will have the same value: the table index of this variable; the second I*2 number in each element will be its index in the array. REC_PTR and REC_SIZE can be accessed once the table index is known. The address of the Nth record element in an IGORE record array is REC_PTR + (N-1) * REC_SIZE. Of course N must with in the range of NREC.

For passing entire record arrays or contiguous subarrays of record arrays, the total size is computed according to (N_LAST - N_FIRST + 1) * REC_SIZE, where N_FIRST and N_LAST are the first and last indeces in the array or subarray (and are in the range of NREC).

3.2 General Terminology

4 IDL

4.1 IDL Variables

4.2 IDL Functions And Procedures

4.3 Linking FORTRAN Applications To IDL

5 STRUCTURES AND RECORDS

The current version of VAX IDL does not provide support for interactive stuctures and records. This section describes the design of IGORE structures and records developed to run under IDL. Most of this section deals with the definition and interactive manipulation of IGORE structures and records. Passing records to applications is discussed in the section on AFEs, although it is mentioned briefly here as well.

5.1 Design Description Overview

First a note concerning terminology. The term Structure refers to the definition of a particular data structure, analogous to a data structure defined within a FORTRAN STRUCTURE/END STRUCTURE block. The term Record refers to the implementation of a particular structure through a declaration, analogous to the FORTRAN RECORD /<Defined_Structure>/ DECLARED_RECORD_NAME statement. IGORE shall provide a means for dynamically defining new structures. The declaration of of records shall also be dynamic; that is, any number of records can be interactively declared, each corresponding to a previously defined structure.

5.1.1 Structure Design

The core of the IGORE structures design is the structure descriptor. All records declared to be of a particular structure type share the same structure descriptor. The structure descriptor for a particular structure type is accessed each time a record of that structure type is being accessed interactively. Structure descriptors are discussed in the next subsection. Dynamic access, interactive definition, and management of structures are discussed in subsequent subsections.

5.1.1.1 Structure Descriptors

The structure descriptor, referred to as STRUC_DESCR, is actually a concatanation of variable-lengh descriptors, one for each of the fields defined in the structure. The first part of each field descriptor is a delimited character string which names the field; the second part each field descriptor describes how the data are organized in the field. Every full structure descriptor is preceded by an eight-byte VAX dynamic string descriptor in order to allow the entire structure descriptor to be accessed as a single character string.

Data in a given field of a record of a particular structure type is referred to as either Direct of Indirect, according to where it is located with respect to the data in the other fields of that record. The Direct portion of a record is a contiguous block of memory containing: 1) the actual data associated with each field; 2) a pointer to the actual data which resides outside the contiguous block of memory; or 3) a data descriptor (DATA_DESCR) for the actual data which resides outside the block of contiguous memory. The Indirect portion is the actual data which is not located in the same contiguous bloc of memory as the Direct portion, but is pointed to by either a pointer or a data descriptor in the Direct portion. Each field in a structure is accessed in one of three primary or default modes. These are:

1. Direct mode. In this mode, the field descriptor describes data located in the Direct portion of any record of the particular structure type. The data are described by their type, and the number and sizes of dimensions, if any.

2. Simple indirect (pointer) mode. In this mode, the field descriptor describes data which is located in an Indirect portion of any record of the particular structure type. The data item located in the Direct portion of the record is the starting address of the associated Indirect portion. The data are described by their type, and the number and sizes of dimensions, if any.

3. Complex indirect (data descriptor) mode. In this mode, the field descriptor simply flags this data field to be associated with a data descriptor, but contains no other useful information about the associated data. The data item located in the Direct portion of the record is a DATA_DESCR which describes the actual data associated with the field, as well as the Direct portion of the record in which it (the data descriptor) is embedded.

   The actual data associated with the DATA_DESCR may either be in memory, in which case the DATA_DESCR contains a pointer to it; or the data may be in a disk file, in which case the DATA_DESCR contains a pointer to a table which may be used to access the disk file and the specific data. Such a table will contain information such as FILENAME and TAGWORD.

All three modes may be combined in a given structure type (and hence all records declared to be of this type). The default access to any field will always be to the actual data associated with the field. It is also possible to access pointers and data descriptors as data entities (e.g., extract all the data descriptors from an array of records -- direct and indirect data -- and construct an array of just data descriptors).

5.1.1.2 Dynamic Access Of Strucutures

Structure descriptors are accessed as character strings. The first eight bytes of every structure descriptor is a VAX dynamic string descriptor. A single utility routine accesses all structure descriptors, reading them into a CHARACTER*(*) variable.

A given field in a given structure descriptor is accessed by its name, as stored in the variable-length name portion of each field descriptor. The FORTRAN INDEX function is used to locate the field name. The field descriptor portion directly follows the field name, and is accessed as a standard FORTRAN record whose fields completely describe the associated data (type, number and size of dimensions, etc.). The field descriptors which follow the field names may be variable in length, even though they are accessed via a standard-length FORTRAN record; one of the fields is the record describes the actual length of the particular field descriptor currently accessed.

5.1.1.3 Interactive Structure Definition

5.1.1.4 Structure Descriptor Table

5.1.1.5 Structure Descriptor Libraries

5.1.1.6 Structure Type Aliases

5.1.2 Record Design

5.1.2.1 Dynamic Record Descriptors

5.1.2.2 Associated Tables

5.1.2.3 Record Aliases

5.1.2.4 Interactive Record Operations

5.2 Modules And PDL

6 APPLICATION FRONTEND (AFE)

A fundamental task of IGORE is to pass and receive data between native IDL variables and native FORTRAN variables in FORTRAN applications. Every application that is to run under IDL shall be interfaced to IDL via an Application Frontend (AFE). Although each application will have its own AFE, the basic actions carried out are the same for all AFEs.

The parameters to IDL function and procedure calls are IDL variables. When the function or procedure is actually a FORTRAN application, the IDL parameters passed in the call are received by the AFE, and each is paired with a FORTRAN variable in the AFE. The FORTRAN variables to which the IDL parameters are paired are in turn the arguments of the application being interfaced (there may be additional arguments of the application call which are not directly paired with IDL parameters to the AFE call).

6.1 Design Description Overview

The design of the AFE is based upon a data structure called the control record, CTRL_REC. Each IDL-FORTRAN variable pair is established via a CTRL_REC; each AFE has an array of CTRL_RECs, one record for each pair. Several parameters in the CTRL_REC determine the specific actions required for the given pair during the general data transfer for the entire collection of pairs in a given AFE.

There are five actions taken on each call to the AFE:

1. Establish the IGORE Condition Handler. This assures that any subsequent errors are handled centrally.

2. Establish the pairing of IDL parameters, passed in the call to the AFE, with their FORTRAN counterparts, which are the parameters of the application being interfaced to IDL via the AFE. Each pair is established in a record defined by the structure CTRL_REC, and a list of these records, one list entry per pair exists in each AFE. On each call to the AFE, the leading ten parameters in each record are initialized; some of these ten are initialize only once, on the first call to the AFE. The list is then passed to AFE routines which check and set the CTRL_REC parameters in preparation for data transfer.

3. Pair checking. The heart of the AFE design is the interpretation and modification of the parameters in each CTRL_REC. These parameters determine the validity of the transfer, make requests for any necessary conversion between data types, set parameters used in actual moving of data between memory locations (when necessary), allocate and free virtual memory, etc. The "intelligence" of the AFE is determined by the CTRL_RECs.

4. Data transfer. Once each CTRL_REC has been set up, data-transfer action can be taken. Two data-transfer actions are taken: one before the call to the FORTRAN application to make any required transfers of data from the IDL variables to their FORTRAN counterparts, and one after the call to make any required reverse transfers. The modes of transfer may include: moving data from one location to another, moving and type-coverting data, creating new main-level IDL variables to receive output, passing pointers of IDL-variable data locations to the FORTRAN application, allocating dynamic memory for execution-time dimensioned record arrays in the FORTRAN application, freeing allocated memory, and redimensioning IDL variables according to possibly redimensioned sizes of their FORTRAN counterparts by the FORTRAN application.

5. Calling the FORTRAN application. The parameters in the call are pointers passed by value (%VAL). The actual values of the pointers are set by the checking routines. A given pointer may be the address of: the actual FORTRAN variable normally used in the call, first byte of an IDL variable's data, or the first byte of dynamically allocated memory. Each of these parameters is associated with an IDL-FORTRAN pair. When ever possible, only pointers to the IDL data are passed to the application and no actual moving of data between memory locations is carried out. Additional arguments of the application may be the dimensions of passed arrays, used for execution-time dimensioning of these arrays in the FORTRAN application and/or for returning new dimension sizes used to redimension the associated IDL variable (if it's an output variable). Passed dimensions may or may not also be explicit parameters in the IDL call to the AFE.

6.1.1 CTRL_REC Structure And Parameter Descriptions

Shown below is a listing of the CTRL_REC_STRUC.ICL include file which defines the structure of the CTRL_REC. The subsections that follow the listing describe each of the parameters in the CTRL_REC.

```
c+++ BEGIN CTRL_REC_STRUC.ICL INCLUDE FILE ++++++++++++++++++++++++++++++

c... This is CTRL_REC_STRUC.ICL file.
c
c    The structure defined here contains pointers, memory
c    allocation requirements, and control flags used by
c    all IGORE AFEs when transferring data between IDL parameters
c    and FORTRAN parameters. Each IDL-FORTRAN pair in a given AFE has
c    associated with it a record of this structure; each AFE has
c    a list of these records, one list entry per pair.
c
c    Upon the first call to the AFE, the first seven parameters in each of
c    the records of the list are initialized. On subsequent calls, only
c    the 8th, 9th, and 10th parameters need to be reset. The list is then
```

```
c    passed to AFE utilities which set/read various other record parameters
c    and ultimately take data-transfer action according to control parameters
c    in each record.  Not all record parameters are applicable to every pair.
c
c    Some default values of parameters which may be used in any AFE are
c    also set here in FORTRAN PARAMETER statements.

c... XFER directions and null array_dims pointer

        byte in
        byte out
        byte in_out
        byte no_xfer_necessary
        integer*4 no_array
        parameter (in=1)
        parameter (out=2)
        parameter (in_out=3)
        parameter (no_xfer_necessary=4)
        parameter (no_array=0)                           !no dims to pass

c... The CTRL_REC structure definition:

        structure /ctrl_rec/
           union
              map
                integer*4       for_var_dptr     !pointer to descr of for_var
                integer*4       for_nam_dptr     !pointer to descr of its name
                integer*4       idl_var_dptr     !pointer to descr of idl_var
                integer*4       array_dims_ptr   !to adjust output idl arrays
                logical*1       req_opt          !required (T) or optional (F)
                logical*1       record_param     !is this a record? T or F
                integer*2       conv_mask        !bits control enabled modes
                logical*1       idl_param_rcvd   !was param passed? T or F
                byte            io_dir           !xfer direction for this pair
                integer*2       conv_mode        !bits control active modes
                integer*2       conv_flag        !bits control action per xfer
                integer*4       for_var_ptr      !pointer fortran variable
                integer*4       idl_var_ptr      !pointer to idl_var's data
                integer*4       param_ptr        !pointer to applic's param
                integer*4       for_var_size     !total bytes in for_var
                integer*4       idl_var_size     !total bytes in idl_var data
                integer*4       nbytes           !bytes when using LIB$GET_VM
                integer*4       nrec             !number of records to pass
                logical*1       for_var_init     !for_var initialized? T or F
              end map
              map
                character*55    pak              !access all at once
              end map
           end union
        end structure

c+++ END CTRL_REC_STRUC.ICL INCLUDE FILE ++++++++++++++++++++++++++++++++
```

6.1.1.1 FOR_VAR_DPTR

This is a pointer (address of) to the VAX descriptor (prototype portion only; i.e., first two longwords) of the FORTRAN variable of the IDL-FORTRAN pair. This parameter is set by the character function BUILD_PAIR, called directly from the AFE.

6.1.1.2 FOR_NAM_DPTR

This is a pointer to the VAX descriptor of the ASCII name of the above FORTRAN variable. For example, if the name of the variable is LIVETIME, the FOR_NAM_DPTR points to a descriptor of the string "LIVETIME". This parameter is set by character the function BUILD_PAIR, called directly from the AFE.

6.1.1.3 IDL_VAR_DPTR

This is a pointer to the IDL variable's descriptor. IDL scalars (excluding strings) have their actual values embedded in their descriptors. IDL arrays have embedded in their descriptors a pointer to a standard VAX array descriptor associated with the actual data. IDL strings have embedded in their descriptors standard VAX dynamic string descriptors (size, type, class, pointer to start of character data). This parameter is set by the character function BUILD_PAIR, called directly from the AFE.

6.1.1.4 ARRAY_DIMS_PTR

This is a pointer to a dimension-size array in the compiled AFE. All subscripted FORTRAN variables in the AFE have associated with them: i) one I*4 variable for each dimension, containing the size of the associated dimension; ii) a dimension-size array containing in its first element the total number of declared dimensions, and in its remaining elements the addresses of the I*4 variables containing the sizes of the dimensions (item i). The pointer to the dimension-size array in the CTRL_REC gives the AFE routines access array dimensions before and after the application call. If ADJUST_OUTPUT_SIZE or CONVERT_ON_OUTPUT is set in CONV_FLAG (see CONV_MODE below), the dimension-size array is used to redimension the IDL array (if it is output). The dimensions of the FORTRAN array must be passed to the FORTRAN application in order to use this capability; i.e., the application must return modified values of the sizes of any arrays for which redimensioning of IDL counterparts is to take place. On every call to the AFE the values of the I*4 dimension-size variables (item i) are reset to their initial values. The value of the pointer is set in the character function character BUILD_PAIR, called directly from the AFE.

6.1.1.5 REQ_OPT

This parameter determines whether or not the supply of the IDL parameter by the main-level IDL call to the AFE is required or optional. Required parameters which are not supplied are prompted for; optional parameters which are not supplied are set to defaults if their FORTRAN counterparts have been initialized (see FOR_VAR_INIT below) or prompted for otherwise. This parameter is set by the character function BUILD_PAIR, called directly from the AFE; its value is set at coding time as a programmer option.

6.1.1.6 RECORD_PARAM

This parameters flags the associated parameter either as a record variable (RECORD_PARAM = TRUE) or a "standard" IDL variable (RECORD_PARAM = FALSE). The value of the flag controls action taken by the checking routines. This parameter is set by the character function BUILD_PAIR, called directly from the AFE; its value is set at coding time as a programmer option.

6.1.1.7 CONV_MASK

This parameter controls which of the conversion modes in CONV_MODE may be enabled and disabled interactively (see CONV_MODE below). Each bit is associated with the same mode as the corresponding bit in CONV_MODE. A bit value of 0 indicates that the associated mode is disabled and may not be enabled interactively within IGORE; the value of the corresponding bit in CONV_MODE is always 0 in this case. A bit value of 1 indicates that the associated mode may be enabled and disabled interactively; the default value of the corresponding bit in CONV_MODE may be 0 or 1.

This parameter is set by the character function BUILD_PAIR, called directly from the AFE; its value is set at coding time as a programmer option.

6.1.1.8 IDL_PARAM_RCVD

This parameter flags whether or not the IDL parameter was actually supplied on a given call to the AFE. The convention for determining the value of this flag is to assume that trailing parameters were not passed if the actual number passed is less than the number expected. No place-holders are allowed. For example, if eight parameters are expected (compiled in the parameter list of the AFE) and only four are passed, IDL_PARAM_RCVD will be set TRUE for the first four parameters, and FALSE for the last four. This parameter is set by the character function BUILD_PAIR, called directly from the AFE.

6.1.1.9 IO_DIR

This parameter controls the direction of transfer for the associated IDL-FORTRAN pair. The possible values are as follows:

1. IO_DIR = 1 ... IDL-to-FORTRAN only. The value of the IDL variable is preserved; the IDL parameter is ignored when data are transferred from the FORTRAN variables after return from the FORTRAN application.

2. IO_DIR = 2 ... FORTRAN-to-IDL only. The IDL parameter receives output only; the IDL parameter need not exist prior to the IDL call to the AFE. The specific actions taken depend on whether or not the IDL parameter already exists when the IDL call to the AFE is made, and on specific settings in the CONV_MODE.

3. IO_DIR = 3 ... IDL-to-FORTRAN and FORTRAN-to-IDL. Data are tranferred between the members of the IDL-FORTRAN pair on both transfers: before and after the call to the FORTRAN application.

4. IO_DIR = 4 ... No tranfer necessary in either direction. This is not an error condition. It is used when the pointer to the actual IDL data (see IDL_VAR_PTR below) is passed to the FORTRAN application or when an optional input-only (IO_DIR = 1) parameter has not been passed but has already been initialized.

5. IO_DIR = 0 ... Illegal transfer request. No transfer will take place, and the AFE will abort before calling the FORTRAN application.

In addition, the values -1, -2, and -3 indicate that some kind of type conversion is necessary on transfers with IO_DIR = 1, 2, or 3, respectively.

This parameter is set by the character function BUILD_PAIR, called directly from the AFE. On every call to the AFE, the value is reset to an initial value of 1, 2, or 3; the value subsequently may be modified by the checking routines. The initial value is supplied at coding time as a programmer option.

6.1.1.10 CONV_MODE

This parameter controls which type conversion modes are enabled for the associated IDL-FORTRAN pair. Each bit controls the enabling/disabling of a unique mode. Bit values of 1 (0) enable (disable) the associated mode. All numeric conversions between one-, two-, four-, and eight-byte data elements are supported (excluding COMPLEX*8). Four additional conversion actions may be flagged: ACCEPT_SMALLER_SOURCE, CONVERT_ON_OUTPUT, ADJUST_OUTPUT_SIZE, and STRG (string-to-string transfers). See the section below on CONVERSION_MNEMONICS for the bit settings and associated conversions, and details on conversion actions.

This parameter is set by the character function BUILD_PAIR, called directly from the AFE; its value is set at coding time as a programmer option and may be reset interactively within IGORE, subject to the value of CONV_MASK (next item).

6.1.1.11 CONV_FLAG

This parameter controls the conversion mode which is to be active for the associated IDL-FORTRAN pair on any given call to the transfer routines (activated by an actual call to an AFE). This parameter is set by the checking routines.

6.1.1.12 FOR_VAR_PTR

This is a pointer to the actual FORTRAN variable compiled in the AFE code. When transfer is required, this pointer is a source and/or a destination. For strings, FOR_VAR_PTR is the address of the string's descriptor. This parameter may also be a reference to dymanically allocated memory. This parameter is set by the checking routines.

6.1.1.13 IDL_VAR_PTR

This is a pointer to the actual data associated with the IDL variable. When transfer is required, this pointer is a source and/or destination. For strings, IDL_VAR_PTR is the address of the string's descriptor. This parameter is set by the checking routines.

6.1.1.14 PARAM_PTR

This parameter is the pointer that is passed by value (%VAL) to the FORTRAN application. It is set in the checking routines. Its value is either FOR_VAR_PTR, IDL_VAR_PTR, or the starting address of dynamically allocated memory returned by a call to LIB$GET_VM.

6.1.1.15 FOR_VAR_SIZE

This parameter is the total number of bytes of data associated with the FORTRAN variable. It is set by the checking routines. Its value is either the compiled size of the FORTRAN variable, or the number of bytes used in a request for dynamic memory allocation.

6.1.1.16 IDL_VAR_SIZE

This parameter is the total number of bytes of data associated with the IDL variable. It is set by the checking routines.

6.1.1.17 NBYTES

This parameter is the total number of bytes used in a request for dynamic memory allocation (when such a request is necessary). It is set by the checking routines.

6.1.1.18 NREC

This parameter is the total number of records in an array of records to be passed to the FORTRAN application. Its value may be either the number of records in the associated IDL record array or a default set in the dimension-size array (see ARRAY_DIMS_PTR above). It is set by the checking routines.

6.1.1.19 FOR_VAR_INIT

This parameter indicates whether or not the FORTRAN counterpart of an optional IDL parameter has been initialized. If an optional parameter is not passed and FOR_VAR_INIT is TRUE, then no transfer takes place (IO_DIR = 4) and PARAM_PTR is set to FOR_VAR_PTR, causing the previous value of the FORTRAN variable to be passed to the FORTRAN application. If FOR_VAR_INIT is FALSE, then the user is prompted to supply the IDL parameter.

6.1.1.20 PAK

This is a character map of the entire structure of the CTRL_REC to enable accessing the record its entirety, and packing its fields with single calls to character functions.

6.1.2 Parameter Classifications

Parameters of all IDL function and procedure calls are IDL native variables. The supported data types are: byte, I*2, I*4, R*4, R*8, COMPLEX*8, and strings. Arrays of up to eight dimensions of any of these types are also supported. In the AFE, the FORTRAN counterparts of the IDL parameters may be any of the corresponding data types or, in addition, a representative of a structured record. The actions taken by the AFE and its called routines depend on how the IDL-FORTRAN pair elements match. A parameter pair is classified as variable type if its FORTRAN element is one of the supported IDL types; or it is classified as a record type if its FORTRAN element is a representative of a VAX structured record. In addition, all parameters are classified as required or optional.

6.1.2.1 Variables

Variables include all string, and one-, two-, four-, and eight-byte numeric type scalars and arrays. The FORTRAN element in each IDL-FORTRAN pair is declared in the AFE with its type and dimensions. The type corresponds to the type expected in the application. String element lengths are declared in the AFE and are passed to the application (i.e., declared in the application as CHARACTER*(*)) or declared with the same length in the AFE and in the application.

For arrays of all types, the dimensions are passed to the application for execution-time dimensioning. Arrays passed as parameters to applications should not be declared with hard-wired dimensions in the application; they must not be declared with hard-wired dimensions if passed dimensions are also passed as parameters to the application.

The memory compiled in the AFE for each declared FORTRAN variable may or may not be used on a given call to the AFE. Usage depends on the transfer direction mode of the variable (see below), whether or not type conversion is required, and whether or not dimensions of arrays are modified by the application (in this case, dimensions may only be reduced in size).

6.1.2.2 Records

Records are declared in the application to be of a type defined in a FORTRAN STRUCTURE statement construction. They must be dimensioned symbolically with a parameter that is passed as an argument to the application. In the AFE, the corresponding records are represented by character strings which are set to string mnemonics for the particular structure types of the declared records in the application. Each structure type has a unique mnemonic.

No static allocation is compiled in the AFE for records. The AFE passes only pointers to the memory locations of records and the dimensions of record arrays to the application. The pointers are addresses either of existing IGORE records, memory allocated dynamically by the AFE routines into which IGORE records are transferred, or memory allocated dynamically by the AFE routines as temporary buffer space for the application. For records designated as OUTPUT ONLY (see the next section), the IGORE record need not exist prior to the call to the AFE. In this case, the AFE must include a default dimension size in order to allocate a default storeage space for the application.

VAX IDL currently does not support records and structures. IGORE does support records and structures, and uses native IDL I*4 variables to represent IGORE records at the IDL command level (see section 4 above). The IDL variables used to represent IGORE records shall be referred to as RECORD ALIASES. As far as IDL is concerned, a record alias is just an I*4 variable. It is possible to give an IGORE record a unique name, but manipulation of records at the IDL command level is accomplished only through record aliases; it is not possible to restrict the number of aliases that a given record may have.

6.1.2.3 Required And Optional Parameters

All parameters in the IDL call to the AFE are additionally classified as require or optional. The designation is determined at AFE coding time as a programmer option. Only INPUT ONLY parameters may be designated as optional. Also, in the argument list of a given AFE, only the trailing parameters in the list may be designated as optional.

Required parameters are those which must be supplied explicitly on each and every call to the AFE. If not supplied, the user will be prompted to supply them. Optional parameters are those which must be initialized on at least one call to the AFE, but need not be supplied explicitly on subsequent calls. The initialization may occur in the statement which declares the FORTRAN element of the IDL-FORTRAN pair in the AFE at compilation time. Explicity supply of optional parameters overrides previous initialized values and updates the initialized values for subsequent calls.

6.1.3 Parameter Transfer Direction Modes

Three basic transfer direction modes are distinguished: input only, output only, and input/output. The mode of a particular parameter is determined at AFE coding time as a programmer option. In certain specific cases, no actual transfer of data between memory locations may be required, and only pointers to the data are passed. This mode of transfer is referred to as NO TRANSFER NECESSARY. All string-type variables, whether scalar or arrays, always require data transfer. Some transfers are not allowed at all and are flagged as ILLEGAL TRANSFERs; no actual moving of data is carried out and the AFE aborts.

6.1.3.1 Input Only

This mode of transfer is flagged with the mnemonic IN; the value is 1. The value of the IDL input parameter is unchanged by these transfer because the application is guaranteed to use its own copy of the data. Specific action depends on whether the parameter is classified as a variable or as a record.

For variable-type parameters, the application is always passed a pointer to the FORTRAN element in the IDL-FORTRAN pair. On a given IDL call to the AFE, actual transfer of data from the input IDL parameter to the FORTRAN counterpart is required or not according to the specific situation:

1. If the parameter is required, then data from the input IDL parameter are always transferred to the FORTRAN counterpart. If the parameter is not supplied explicitly in the IDL call to the AFE, the user is prompted to supply it.

2. If the parameter is optional and supplied explicitly in the call, then data from the input IDL parameter are transferred to the FORTRAN counterpart.

3. If the parameter is optional and not supplied explicitly in the call and the FORTRAN counterpart has previously been initialize, then no tranfer is made; the transfer is flagged NO TRANSFER NECESSARY.

4. If the parameter is optional and not supplied explicitly in the call and the FORTRAN counterpart has not previously been initialize, then the user is prompted to supply it.

5. If data are to be transferred and type conversion is required and allowed by the CONV_MODE in the CTRL_REC, IN is set to -IN.

For record-type parameters, the application is always passed a pointer to the dynamically allocated memory contain a copy of the input IGORE record and the number of record elements therein. On a given IDL call to the AFE, actual transfer of the IGORE record to this memory is required or not depending on the specific situation:

1. If the parameter is required, then the requisite memory is allocated and the input IGORE record (direct and indirect portions) is copied into this space. If the parameter is not explicitly supplied, then the user is prompted to supply it.

2. If the parameter is optional and explicitly supplied, then any previous memory allocation containing initialized data is freed, the requisite memory for the newly-supplied record is allocated and the input IGORE record (direct and indirect portions) is copied into this space.

3. If the parameter is optional and not supplied explicitly in the call and the FORTRAN counterpart has previously been initialize, then no tranfer is made; the transfer is flagged NO TRANSFER NECESSARY.

4. If the parameter is optional and not supplied explicitly in the call and the FORTRAN counterpart has not previously been initialize, then the user is prompted to supply it.

6.1.3.2 Output Only

This transfer mode is flagged with the mnemonic OUT; the value is 2. Output only parameters need not exist at the time of a given IDL call to the AFE. The specific actions allowed and/or taken depend on whether the parameter is a variable or a record, and whether it exists prior to the call to the AFE.

For variable-type parameters the possible actions are as follows:

1. If the IDL variable does not exist, a pointer to the FORTRAN element of the IDL-FORTRAN pair will be passed to the application along with the default dimensions (if any) for that variable. The AFE routines will create an appropriate IDL variable during the output transfer after the application has returned to the AFE; the name of the variable will be that of the explicitly passed but undefined parameter. Actual transfer of data will take place.

2. If the IDL variable does exist and is of the same type as the FORTRAN counterpart and the dimension-mapping rules are satisfied (see Transfer of Dimensions below), a pointer to the IDL data and its dimensions (if any) will be passed to the application. If the dimensions are not modified by the application, no transfer of data will take place. If the dimensions are modified (reduction in size is the only allowable modification), the IDL variable is recreated to

have the new dimensions and the data are transferred to the redimensioned variable.

3. If the IDL variable does exist but is of different type than the FORTRAN counterpart, the case reverts to that of an undefined IDL variable as long as the CONV_MODE allows for CONVERT_ON_OUTPUT (see Conversion). Otherwise, an ILLEGAL TRANSFER is flagged.

For record-type parameters the possible actions are as follows:

1. If the record does not exist, a default dimension in the AFE is used to dynamically allocate memory and a pointer to this memory, along with the default dimension, is passed to the application. The transfer direction mode is set to -OUT. Upon return from the application to the AFE, a new IGORE record is created and name of the explicitly-passed parameter is used as the record alias. Actual data transfer takes place in this case. The application may return a smaller number of records than the default supplied by the AFE. In this case, if CTRL$ADJUST_OUTPUT_SIZE is set (see Conversions below) the newly-created IGORE record array will be the corrected size, and not the size of the default.

   The memory allocated by the AFE routines for the direct portion of the record(s) is released after the IGORE record(s) is (are) created and the data in the dynamic memory transferred. Any indirect portions of the record(s) do not get transfer; their pointers are simply inherited by the new IGORE record.

2. If the record does exist and is of the same type as that in the application (as determined by the FORTRAN string mnemonic in the AFE), a pointer to the IGORE record along with the dimension is passed to the application. No actual transfer of data takes place. No redimensioning is allowed in this case because the existing record may be intermediate elements of a larger record array.

3. If the record does exist but is of a different type than that in the application, the transfer is flagged as ILLEGAL TRANSFER.

6.1.3.3 Input/Output

This transfer mode is flagged with the mnemonic OUT; the value is 2. The IDL variable must exist prior to the call to the AFE. The specific actions allowed and/or taken depend on whether the parameter is a variable or a record, and whether it exists prior to the call to the AFE.

For variable-type parameters the possible actions are as follows:

1. If the IDL parameter is of the same type as its FORTRAN counterpart and the dimension-mapping rules are satisfied (see Transfer of Dimensions below), a pointer to the IDL data along with its dimensions are passed to the application. No actual transfer of data takes place and no redimensioning is ever allowed.

2. If the IDL parameter is of a different type than its FORTRAN counterpart, then number of elements in both must be exactly the same in addition to the contraint that the dimension-mapping rules are satisfied, otherwise an ILLEGAL TRANSFER is flagged. If these conditions are met, then the input IDL parameters are flagged for the appropriate conversion if allowed by the CONV_MODE; if not allowed an ILLEGAL TRANSFER is flagged. Actual data transfer takes place. On the input transfer, the data are converted from the IDL type to the FORTRAN type. On the output transfer one and only one of the remaining two items is carried out.

3. If CONVERT_ON_OUTPUT is set, then the IDL variable is converted to the type of its FORTRAN counterpart. Redimensioning of the number of elements is never allowed. Actual data transfer takes place.

4. If CONVERT_ON_OUTPUT is not set, then the FORTRAN data are converted back to the type of the original IDL variable. Redimensioning of the number of elements is never allowed. Actual data transfer takes place.

For record-type parameters the possible actions are as follows:

1. If the record does not exist, the user is prompted to supply an exisiting record.

2. If the record does exist and is of the same type as that in the application (as determined by the FORTRAN string mnemonic in the AFE), a pointer to the IGORE record along with the dimension is passed to the application. No actual transfer of data takes place. No redimensioning is allowed in this case because the existing record may be intermediate elements of a larger record array.

3. If the record does exist but is of a different type than that in the application, the transfer is flagged as ILLEGAL TRANSFER.

6.1.3.4 No Transfer Necessary

This transfer mode is flagged with the mnemonic NO_XFER_NECESSARY; the value is 4. It is used when no actual move of data between memory locations is required, and only a pointer to the source data is passed to the application. This mode is never flagged for string variables; passing strings always involves moving data between memory locations. The conditions for NO_XFER_NECESSARY are the following:

1. For input only parameters which are optional for which the FORTRAN element of the IDL-FORTRAN pair has been initialized. The pointer to the FORTRAN variable is passed to the application, along with the dimensions (if any). This condition holds for variable-type and record-type parameters.

2. For output only and input/output variable-type parameters, if the types of the IDL-FORTRAN pair elements are the same and the dimension-mapping (see Transfer of Dimensions below) rules are satisfied, a pointer to the IDL data is passed to the application, along with the dimensions (if any).

3. For output only and input/output record-type parameters, if the IGORE record exists and is of the same type as the record expected by the application, a pointer to the IGORE record data is passed to the application, along with the dimensions (if any).

6.1.3.5 Illegal Transfers

This transfer is flagged with a 0. The AFE routines should not get as far as the transfer utilities if any of the parameter pairs are flagged this way. If the data transfer routine does manage to get called with any of the parameters flagged as illegal transfers, the routine will abort. Conditions that cause this flag to be set are:

1. Source size larger than desination size for a requested data move.

2. Conversion request that is not supported by IGORE or the current setting of the CONV_MODE.

6.1.4 Transfer Of Dimensions

When either or both of the elements in an IDL-FORTRAN pair is an array, the dimensions -- number and sizes -- of the IDL array may have to be transferred to the AFE and passed on to the application. The conditions under which transfer is required and the rules governing such transfers are given here.

6.1.4.1 Conditions Requiring Dimensions Transfer

A necessary condition for which dimension transfer may be required is that either or both elements of the IDL-FORTRAN pair is an array. Any one of the remaining conditions listed below in addition makes transfer of dimensions required. The remaining conditions are:

1. All input only parameter transfers.

2. Output only and input/output variable-type parameter transfers for which the data types of the IDL and the FORTRAN variables match.

3. Output only and input/output record-type parameter transfers for which the record types of the IGORE and the FORTRAN parameters match.

Note that the requirement for dimension transfer does not guarantee that the rules for dimension mapping are satisfied. These are given in the next subsection.

6.1.4.2 Dimension-Mapping Rules

Dimension mapping always refers to mapping dimensions of the IDL variable passed to the AFE as a parameter to the FORTRAN variable in the AFE. If the rules are satisfied for a given transfer, the transfer is made; otherwise an error is signalled and the program aborts. The rules for dimension mapping are as follows:

1. If the number of dimensions in both the IDL and FORTRAN variables are the same, the values of the IDL dimensions are transferred to the corresponding FORTRAN dimensions. This is referred to as one-to-one mapping.

2. If the IDL variable is a scalar and the FORTRAN variable has a single dimension, the value of the single FORTRAN dimension is set to 1.

3. If the IDL variable is multiply-dimensioned and the FORTRAN variable is singly-dimensioned, the value of the single FORTRAN dimension is set to the total number of elements in the IDL variable; i.e., the product of the IDL dimensions. This is referred to as many-to-one mapping.

Note that there is no one-to-many mapping or many-to-differentmany mapping.

6.1.5 Conversion And CONVERSION_MNEMONICS

IGORE shall support certain types of conversions between the IDL and FORTRAN elements of IDL-FORTRAN pairs. The combination of supported conversions applicable to a given pair is determined by the CONV_MASK and the CONV_MODE in the CTRL_REC; the setting of these parameters is a programmer option. CONV_MASK is a compiled (static) code for the conversions chosen by the programmer (from among those supported by IGORE) to be applicable in general for a given pair. CONV_MODE is essentially equivalent to CONV_MASK except that it is dynamic within the constraints of CONV_MASK. That is, a particular conversion in the CONV_MODE may be toggled (enabled/diabled) interactively as long as that conversion is (statically) enabled in the CONV_MASK.

The specific conversion(s) required on a given transfer will be determined by the CONV_FLAG in the CTRL_REC. When a type or size mismatch of pair elements is detected in the pair-checking routines of the AFE, the routines attempt construct a conversion request with the mnemonic CONV_RQST which, if carried out, will remedy the mismatch. If no CONV_RQST can be constructed to remedy the mismatch, the specific transfer is flagged illegal. If a CONV_RQST can be constructed, it is compared with the CONV_MODE to determine if the requested conversion is permitted for this particular pair. If it is allowed, the CONV_FLAG is set to the CONV_RQST and at transfer time the conversion specified in the CONV_FLAG is carried out. If not allowed, the specific transfer is flagged illegal.

Conversions are classified as numeric or as special action. In either case, a specific conversion is represented by a unique mnemonic; the value of the mnemonics are given in the next subsection. In general, the complete CONV_MODE can be built by adding together the mnemonics for each conversion; for numeric conversion, the conversion direction is set in the high order bit, so both directions are represented by setting the bit for the unidirectional conversion and also setting the high order bit. Note that in all cases in which conversion is carried out, data have to be moved between memory locations.

6.1.5.1 Numeric Conversions

IGORE supports numeric conversions between all one-, two-, four-, and eight-byte data types (excluding strings and COMPLEX*8). The conversions are done with MACRO conversion instructions whose arguments are the memory addresses of the source and destination data elements, the number of elements to convert, and a MACRO mnemonic specifying the type of conversion required.

The IGORE mnemomics for numeric conversion are defined by setting one bit in the CONV_MODE to specify the data types of the two elements involved in the conversion, and by a one or zero in the high order bit (bit 15) to specify the direction (source and destination) of the conversion. All possible numeric conversions are set in CONV_MODE by adding all the mnemonics for the unidirectional conversion and setting bit 15 to specify bidirectionality for all those conversions.

The mnemonics for numeric conversions are given in the next section along with the special action conversions described in the following section. Note that numeric conversions have no meaning for record-type parameters.

6.1.5.2 CONVERSION_MNEMONICS Matrix

When a type mismatch is detected for a pair, a conversion request must be constructed by selecting an appropriate conversion from among the conversions supported by IGORE. The selection of the appropriate numeric conversion mnemonic for a given type mismatch makes use of a square matrix whose rows correspond to the IDL type, whose columns correspond to the FORTRAN type, and whose elements are the mnemonics. Each conversion mnemonic is connected to its inverse by reflection about the diagonal of the matrix.

Mismatched pairs for which no conversion is allowed are designated with the mnemonic NOOP, which will translate into an illegal transfer. Diagonal elements of the matrix connect like types and, except for string types, are also designated NOOP. The diagonal element connecting two string types has the mnemonic STRG, signifying string passing; string passing is handled as a special case. Except for string-string "conversions," the conversion matrix is never accessed for like types; illegal transfer is not flagged on the basis of the matrix element NOOP for like types.

A simple one-statement algorithm is utilized that translates the type code of any given parameter into its row or column index. The algorithm is used twice for a given pair, once for the IDL parameter and once for the FORTRAN parameter. The two resulting indeces specify the mnemonic for conversion between the pair elements.

The CONVERSION_MNEMONIC include file is listed below. The file includes the conversion mnemonics and the CONVERION_MNEMONICS matrix.

```
c+++ BEGIN CONVERSION_MNEMONICS.ICL INCLUDE FILE ++++++++++++++++++++++++++

c... CONVERSION_MNEMONICS.ICL include file.
c    This include file sets the values of the symbolic mnemonics which
c    correspond to distinct conversion modes in IGORE.  Each value is
c    an I*2 number with a single bit set.  For the numeric type conversions,
c    the mnemonic represents the conversion from the type of the first
c    element to the type of the second element.  When bit 15 is set,
c    the mnemonic represents the inverse conversion.
c
c    A matrix of conversion mnemonics for numeric type conversion is set up
c    in the 8x8 array CONV_MATRIX.  The values reflected about the diagonal
c    elements differ only in the parity of bit 15.
c    Proper choice of indeces of the source and destination variables will
c    result in the correct mnemonic for the convesion from source type
c    to destination type.

c... Declare the mnemonics to be I*2
```

```
      integer*2 NOOP                              !no conversion
      integer*2 CTRL$ILLEGAL_CONVERSION           !flags illegal conversion
      integer*2 B15                               !bit 15 only
      integer*2 R8I4                              !R*8 --> I*4 conversion
      integer*2 R8I2                              !R*8 --> I*2 conversion
      integer*2 R8R4                              !R*8 --> R*4 conversion
      integer*2 R4I4                              !R*4 --> I*4 conversion
      integer*2 R4I2                              !R*4 --> I*2 conversion
      integer*2 I4I2                              !I*4 --> I*2 conversion
      integer*2 I4R8                              !I*4 --> R*8 conversion
      integer*2 I2R8                              !I*2 --> R*8 conversion
      integer*2 R4R8                              !R*4 --> R*8 conversion
      integer*2 I4R4                              !I*4 --> R*4 conversion
      integer*2 I2R4                              !I*2 --> R*4 conversion
      integer*2 I2I4                              !I*2 --> I*4 conversion
      integer*2 B1I2                              !B*1 --> I*2 conversion
      integer*2 B1I4                              !B*1 --> I*4 conversion
      integer*2 B1R4                              !B*1 --> R*4 conversion
      integer*2 B1R8                              !B*1 --> R*8 conversion
      integer*2 I2B1                              !I*2 --> B*1 conversion
      integer*2 I4B1                              !I*4 --> B*1 conversion
      integer*2 R4B1                              !R*4 --> B*1 conversion
      integer*2 R8B1                              !R*8 --> B*1 conversion
      integer*2 STRG                              !flag string variables
      integer*2 CTRL$CONVERT_ON_OUTPUT            !convert IDL to FORTRAN type
      integer*2 CTRL$ADJUST_OUTPUT_SIZE           !resize IDL output
      integer*2 CTRL$ACCEPT_SMALLER_SOURCE        !input IDL size lt FOR size
      integer*2 NUM_CONV_MASK                     !preserves bits (0-5,15) only

c... Set the values (bits)

      parameter(NOOP=0)                           !no bits set
      parameter(CTRL$ILLEGAL_CONVERSION=0)        !no bits set
      parameter(B15=-32768)                       !bit 15 set
      parameter(R8I4=1)                           !bit 0 set
      parameter(R8I2=2)                           !bit 1 set
      parameter(R8R4=4)                           !bit 2 set
      parameter(R4I4=8)                           !bit 3 set
      parameter(R4I2=16)                          !bit 4 set
      parameter(I4I2=32)                          !bit 5 set
      parameter(I4R8=R8I4+B15)                    !bits 0 and 15 set
      parameter(I2R8=R8I2+B15)                    !bits 1 and 15 set
      parameter(R4R8=R8R4+B15)                    !bits 2 and 15 set
      parameter(I4R4=R4I4+B15)                    !bits 3 and 15 set
      parameter(I2R4=R4I2+B15)                    !bits 4 and 15 set
      parameter(I2I4=I4I2+B15)                    !bits 5 and 15 set
      parameter(B1I2=512)                         !bit 9 set
      parameter(B1I4=1024)                        !bit 10 set
      parameter(B1R4=2048)                        !bit 11 set
      parameter(B1R8=4096)                        !bit 12 set
      parameter(I2B1=B1I2+B15)                    !bits 9 and 15 set
      parameter(I4B1=B1I4+B15)                    !bits 10 and 15 set
      parameter(R4B1=B1R4+B15)                    !bits 11 and 15 set
      parameter(R8B1=B1R8+B15)                    !bits 12 and 15 set
```

```
      parameter(STRG=8192)                       !bit 13 set
      parameter(CTRL$CONVERT_ON_OUTPUT=64)       !bit 6 set
      parameter(CTRL$ADJUST_OUTPUT_SIZE=128)     !bit 7 set
      parameter(CTRL$ACCEPT_SMALLER_SOURCE=256)  !bit 8 set
      parameter(NUM_CONV_MASK=R8I4+R8I2+R8R4+R4I4
     1                       +R4I2+I4I2+B1I2+B1I4
     2                       +B1R4+B1R8+STRG+B15)   !bits 0-5,9-13,15 set

c... Declare and initialize CONV_MATRIX

      integer*2 conv_matrix(8,8)

      data conv_matrix /NOOP,NOOP,NOOP,NOOP,NOOP,NOOP,NOOP,NOOP,
     1                  NOOP,NOOP,I2B1,I4B1,R4B1,R8B1,NOOP,NOOP,
     2                  NOOP,B1I2,NOOP,I4I2,R4I2,R8I2,NOOP,NOOP,
     3                  NOOP,B1I4,I2I4,NOOP,R4I4,R8I4,NOOP,NOOP,
     4                  NOOP,B1R4,I2R4,I4R4,NOOP,R8R4,NOOP,NOOP,
     5                  NOOP,B1R8,I2R8,I4R8,R4R8,NOOP,NOOP,NOOP,
     6                  NOOP,NOOP,NOOP,NOOP,NOOP,NOOP,NOOP,NOOP,
     7                  NOOP,NOOP,NOOP,NOOP,NOOP,NOOP,NOOP,STRG/

c... That's all

c+++ END CONVERSION_MNEMONICS.ICL INCLUDE FILE +++++++++++++++++++++++++++++
```

6.1.5.3 Special Actions

Four special action conversion mnemonics may also be included in CONV_MASK, CONV_MODE, and CONV_FLAG. These are described below.

1. STRG. This specifies that both elements in the IDLFORTRAN pair are strings. STRG does not signify an actual conversion, but simply directs the transfer routines to execute the special handling required for passing strings.

2. CTRL$CONVERT_ON_OUTPUT. This specifies the action required on the output transfer after the application has returned to the AFE. On input, the IDL the data contained in parameter is always converted to the type of its FORTRAN counterpart (within the constraints of CONV_MODE). On output, two possible actions are possible: 1) convert the FORTRAN data back to original type in of the IDL counterpart; or 2) convert the actual IDL parameter to the type of its FORTRAN counterpart. When CTRL$CONVERT_ON_OUTPUT is clear (not set), the first option is used; when CTRL$CONVERT_ON_OUTPUT is set, the second option is used. Setting CTRL$CONVERT_ON_OUTPUT will also cause the dimensions of the FORTRAN variable to be assigned to the converted IDL variable. This converion action is only applicable to variable-type parameters; it has no meaning for record-type parameters.

3. CTRL$ADJUST_OUTPUT_SIZE. This specifies the action required on the output transfer after the application has returned to the AFE. If the types of the two elements in the IDLFORTRAN pair are the same, but the dimensions are modified by the application, then CTRL$ADJUST_OUTPUT_SIZE specifies that the output IDL parameter should be redimensioned according to the new dimensions returned from the application. This conversion mode is permitted only for OUTPUT-ONLY parameters; that is, INPUT/OUTPUT parameters must maintain the same number of data elements before and after the call to the application. For variable-type parameters this mode is equivalent to CTRL$CONVERT_ON_OUTPUT because it requires redefining the output IDL parameter. This mode may be used for OUTPUT-ONLY records in the special case where the output record (or record array) did not exist prior to the call to the AFE (see Ouput Only Records above).

4. CTRL$ACCEPT_SMALLER_SOURCE. This specifies the action required on the input transfer before the call to the application. If the IDL element of an input only pair is smaller than the compiled size of its FORTRAN counterpart, CTRL$ACCEPT_SMALLER_SOURCE specifies that the transfer may take place (provided the transfer does not violate any conversion rules). In checking the total size of the input IDL parameter, the AFE routines use the number of elements in the IDL parameter, but the size per data element used is that of the FORTRAN variable. This is because the IDL data may first be converted before being transferred to the FORTRAN variables address space. This mode should always be enabled for passing strings unless it is imperative to match the string sizes of the IDL and FORTRAN elements exactly.

6.2 Modules And PDL

There are 15 fundamental modules which make up the AFE routines, as well as several routines which perform simple functions. In addition, there are a number of calls made to other IGORE routines which are not specific to parameter passing. The fundamental modules are listed below, each with a brief description its purpose. The subsequent subsections give the PDL for each module along with a list of other modules called.

1. BLD_PAIR. CHARACTER*20 function. Establishes the IDL-FORTRAN pair in CTRL_REC, initializes static (with noted exceptions) parameters in CTRL_REC.

2. GET_VAX_DESCR. CHARACTER*8 function. This function receives the address of a VAX_DESCR an returns the prototype VAX_DESCR as a character string and number of array elements if variable is an array.

3. GET_IDL_DESCR. CHARACTER*28 function. This function receives the address of an IDL_DESCR and returns the standard IDL_DESCR as character string, along with pointer to data, length of data element(s), number of array elements if IDL variable is an array, number and sizes of array dimensions. All IGORE system routines use the current standard IDL_DESCR as returned by this function. Future modifications to the IDL descriptor shall be accomodated in IGORE by modifying GET_IDL_DESCR to translate the modified IDL descriptor into the current standard IDL_DESCR. If no IDL_DESCR is found, the data pointer, and array dimension number and sizes are all set to zero.

4. CHECK_PAIRS. Subroutine called from the AFE. This routine receives the list of CTRL_RECs set up in the AFE by BLD_PAIR. It loops over each record performing all necessary checks and setting appropriate parameters in each CTRL_REC in preparation for the transfer routines called later from the AFE. CHECK_PAIRS returns an array of status longwords, one for each pair, and a single status longword signalling success or failure. Failure causes the AFE to signal AOE and to pass the status array to the ICH.

5. CRACK_DESCR. INTEGER*4 function. This function receives a single variable-type CTRL_REC and cracks the VAX_DESCR and IDL_DESCR associated with the record. GET_VAX_DESCR and GET_IDL_DESCR are used to crack the associated descriptors. Several other parameters in the CTRL_REC are set according to the pointers, array sizes, etc. returned by GET_VAX_DESCR and GET_IDL_DESCR. The return valueof CRACK_DESCR is a status longword signalling success or IDL_DESCR_NOT_FOUND.

   CRACK_DESCR contains two alternate entry points, SETUP_XFER and TYP_SIZ_CHK (see next two items). The entire module (main and alernate entries) are set up to be reentrant through the main entry point (CRACK_DESCR) and through TYP_SIZ_CHK, by defining dummy functions which simply call CRACK_DESCR and TYP_SIZ_CHK. The reentrant calling sequence is: SETUP_XFER calls TYP_SIZ_CHK (through one dummy function), and TYP_SIZ_CHK calls CRACK_DESCR (though the other dummy function).

6. SETUP_XFER. INTEGER*4 function; alternate entry point in CRACK_DESCR (see CRACK_DESCR above). This function receives a single CTRL_REC then calls the core checking routine TYP_SIZ_CHK, passing it the CTRL_REC. SETUP_XFER then sets the appropriate pointers, data transfer size, CONV_FLAG, and transfer direction mode (including possibly illegal transfer), according to the status and the CONV_RQST return by TYP_SIZ_CHK and on the CONV_MODE. The return value of SETUP_XFER is a status longword signalling success or failure.

7. TYP_SIZ_CHK. INTEGER*4 function; alternate entry point in CRACK_DESCR (see CRACK_DESCR above). This function is the core checking routine for variable-type parameters. It receives a single CTRL_REC then calls CRACK_DESCR (through a dummy function; see above), which cracks the descriptors of the pair elements and initializes various CTRL_REC parameters. TYP_SIZ_CHK checks the sizes and types of the pair elements, determines if conversion is required to remedy any mismatches, then attempts to contruct an appropriate CONV_RQST. The return value of TYP_SIZ_CHK is a status longword signalling complete match, fixable mismatch (valid CONV_RQST), or illegal mismatch.

8. XFER_DIMS. INTEGER*4 function. This function receives the dimension size array associated with the FORTRAN variable (via ARRAY_DIMS_PTR passed by value) and the correponding dimension sizes of the array associated with the IDL parameter. If the dimension-mapping rules are satisfied, the IDL dimensions are transferred to the FORTRAN dimension size array. If not, an error condition is set. The return value of XFER_DIMS is a status longword signalling success or failure.

9. RECORD_PASSING. INTEGER*4 function. This function receives a single record-type CTRL_REC and takes the initial steps in setting up record transfers. The IDL_DESCR is cracked with a call to GET_IDL_DESCR and the pointer value(s) in the associated record alias is used to access the DYNAM_REC_TABLE. If the IGORE record exists, array subranges and memory requirements are determine. If the IGORE record does not exist, the default dimension in the AFE is used to determine memory requirements for output only parameters; or an error is flagged for input only or input/output parameters (these types must exist). If no errors have been encountered to this point, the preparation for record passing is completed by calling RECORD_SETUP (next item). The return value of RECORD_PASSING is a status longword signalling success or failure.

10. RECORD_SETUP. INTEGER*4 function. This function receives a single record-type CTRL_REC and completes the preparation for record passing initiated by RECORD_PASSING. The actions depend on whether the parameter is input only, exiting output only, nonexisting output only, or input/output.

    For input only, any previously allocated memory is freed, new memory is allocated for the new incoming record, and the appropriate pointers, memory size, and transfer direction mode are set. For nonexisting output only, the requisite memory is allocated and the appropriate pointers, memory size, and transfer direction mode are set. For input/output or existing output only, the PARAM_PTR in CTRL_REC is set to the IGORE record address and the dimension NREC is set to that associated with the IGORE record.

The return value of RECORD_SETUP is a status longword signalling success or failure.

11. XFER_PARAMS. Subroutine called from the AFE. This is the main transfer routine for both variable-type and record-type parameters. It receives a transfer mode and the list of CTRL_RECs. The transfer mode specifies the mode in which XFER_PARAMS is called, not the transfer direction mode of any given parameter pair (IO_DIR in CTRL_REC). The two possible tranfer modes of XFER_PARAMS shall be referred to as INPUT_XFER for the call from the AFE prior to the call to the application, and OUTPUT_XFER for the call from the AFE after the call to the application. INPUT_XFER mode gets the IDL data to the application; OUTPUT_XFER gets the return data from the application to the IDL variables which are the parameters in the IDL call to the AFE.

    XFER_PARAMS loops over the entire list of CTRL_RECs taking any appropriate action on each parameter pair associated with eact CTRL_REC. Within the loop over records, six action segments, referred to as Tranfer Branches, are distinguished. On a given pass through the loop, only one transfer branch is executed, depending on the transfer direction mode for the specific pair (IO_DIR) and the transfer mode in which XFER_PARAMS was called (INPUT_XFER or OUTPUT_XFER).

    The transfer branches are: 1) illegal transfer; 2) INPUT_XFER mode and input only or input/output parameters (variable-type and record-type; 3) OUTPUT_XFER mode and output only or input/output variable-type parameters; 4) OUTPUT_XFER mode and output only or input/output record-type parameters; 5) OUTPUT_XFER mode and NO_XFER_NECESSARY parameter transfer direction mode (variable-type and record-type); and 6) INPUT_XFER mode and NO_XFER_NECESSARY parameter transfer direction mode (variable-type and record-type). The action carried out by each of these is described in the PDL section for XFER_PARAMS.

    XFER_PARAMS returns a single status longword signalling success or failure; failure causes the AFE to abort.

12. MOVIT. INTEGER*4 function. This function receives a source and destination address and a source and destination total size (in bytes), then moves the source data to the destination address, provided the destination size is sufficiently large to hold all the source data. The routine calls the VMS RTL routine LIB$MOVC3 (repeatedly if necessary) to move the data. The return value of MOVIT is a status longword signalling success or failure.

13. CONVERT_AND_MOVE. INTEGER*4 function. This function receives a source and destination address, a source and destination total size (in bytes), and a CONVERSION_MNEMONIC. If the size of the source data converted to the destination

data type (as specified by the supplied CONVERSION_MNEMONIC) exceeds the destination size, then and error is flagged and the routine returns to the caller. If no error condition is detected and a numeric conversion is being requested, the CONVERSION_MNEMONIC is translated into a conversion code and passed, along with the source and destination addresses and number of source elements (less than or equal to the total number of source bytes) to the CONVERT function (next item). If the CONVERSION_MNEMONIC is STRG, for string passing, MOVIT is called using the source and destination addresses and sizes in the respective string descriptors. The return value of CONVERT_AND_MOVE is a status longword signalling success or failure.

14. CONVERT. INTEGER*4 function, written in VAX MACRO. This routine receives a source and destination address, a total number of elements, and a conversion code. If the conversion code is recognized, the appropriate VAX MACRO instruction for the requested conversion is executed in a loop over the total number of elements. The return value of CONVERT is a status longword signalling success or failure.

15. SET_DIMBLK. INTEGER*4 function. This routine receives the dimension-size array for a FORTRAN variable array (passed via ARRAY_DIMS_PTR by value) and sets the corresponding array for the IDL array in the return dimension-size array called DIMBLK. The routine also returns the total number of bytes in the FORTRAN variable array. The return value of SET_DIMBLK is a status longword signalling success or failure.

6.2.1 BLD_PAIR

6.2.1.1 Modules Called

6.2.1.2 PDL

6.2.2 GET_VAX_DESCR

6.2.2.1 Modules Called

6.2.2.2 PDL

6.2.3 GET_IDL_DESCR

6.2.3.1 Modules Called

6.2.3.2 PDL

6.2.4 CHECK_PAIRS

6.2.4.1 Modules Called

6.2.4.2 PDL

6.2.5 CRACK_DESCR

6.2.5.1 Modules Called

6.2.5.2 PDL

6.2.6 SETUP_XFER

6.2.6.1 Modules Called

6.2.6.2 PDL

6.2.7 TYP_SIZ_CHK

6.2.7.1 Modules Called

6.2.7.2 PDL

6.2.8 XFER_DIMS

6.2.8.1 Modules Called

6.2.8.2 PDL

6.2.9 RECORD_PASSING

6.2.9.1 Modules Called

6.2.9.2 PDL

6.2.10 RECORD_SETUP

6.2.10.1 Modules Called

6.2.10.2 PDL

6.2.11 XFER_PARAMS

6.2.11.1 Modules Called

6.2.11.2 PDL

6.2.12 MOVIT

6.2.12.1 Modules Called

6.2.12.2 PDL

6.2.13 CONVERT_AND_MOVE

6.2.13.1 Modules Called

6.2.13.2 PDL

6.2.14 CONVERT

6.2.14.1 Modules Called

6.2.14.2 PDL

6.2.15 SET_DIMBLK

6.2.15.1 Modules Called

6.2.15.2 PDL

7 GENERAL TABLES FACILITY

IGORE's operation relies on various tables of information. A central table utility manages all of IGORE's tables, keeping track of the addresses of each tables extensions and the current entry count in each table, allocating memory when new extensions are needed, etc. A directory table provides access to other IGORE tables; it is described in the next subsection. The subsequent subsections describe each of the other IGORE tables.

7.1 Design Description Overview

7.1.1 Directory Table: DIRECTORY_TABLE

This shall refer to the directory of all IGORE tables. The entries in this table can be accessed with the following structure:

```
structure /directory_table/
   character*15 table_type      !mnemonic for table type
   byte         nchar           !actual length of mnemonic
   integer*2    max_entries     !max no. entries per table extension
   integer*2    entry_size      !no. bytes per table entry
   integer*2    current_count   !current number of entries
   integer*2    max_ex          !max no. table extensions
   integer*4    extn_ptr(max_ex)!addresses of each extension
end structure
```

Each table of a given TABLE_TYPE can accomodate MAX_ENTRIES per table extension, with up to MAX_EX extensions allocated dynamically by the central table-managing utilities. MAX_ENTIRES may vary from table type to table type; MAX_EX is the same for all table types; both are hardwired numbers. The maximum number of entries allowed for a given table type is MAX_ENTRIES * MAX_EX. The address of the Nth entry obtained by locating the address of the appropriate table extension and offsetting (positively) the correct number of entries from this address.

7.1.2 Structure Descriptor Pointer Table: STRUC_DESCR_PTR_TABLE

This table locates STRUC_DESCRs of any given type, providing a pointer to the STRUC_DESCR as well as other pertinant information. When access to a particular STRUC_DESCR is required, this table is searched for an entry associated with the mnemonic of the structure type. If an entry is found, it is returned to the calling routine (which presumably knows how to interpret the information in the entry). If no entry is found, IGORE searches its disk-based libraries of STRUC_DESCRs for the given type, loads the associated STRUC_DESCR into memory, and creates an entry in the STRUC_DESCR_PTR_TABLE. If the structure type cannot be found as a table entry or in a library, an error message is issued and an Abort On Error condition is signaled.

IGORE attempts to locate any STRUC_DESCRs not already loaded into memory by sequentially searching two libraries: the user's library and an IGORE system library. The search begins in the user's library.

Once a the STRUC_DESCR for a particular type is loaded into memory, it remains resident for the remainder of the session. IGORE system STRUC_DESCRs reside in readonly shared global commons, but are also loaded only on first refernce. In the case where a system STRUC_DESCR has already been loaded into the shared common, and the user wishes to override its definition without using a different mnemonic for the structure type, it will be possible to redefine the STRUC_DESCR_PTR_TABLE entry to point to the user's STRUC_DESCR.

Each entry in the STRUC_DESCR_PTR_TABLE can be accessed with the following structure:

```
structure /struc_descr_ptr/
   character*15        struc_type      !mnemonic for type
   union
      map
        integer*4      pointer         !address of STRUC_DESCR
        integer*4      length          !no. bytes in STRUC_DESCR
        integer*4      rec_size        !no. bytes per record
      end map
      map
        character*12   pak             !access the rest at one shot
      end map
   end union
end structure
```

Each search of the STRUC_DESCR_PTR_TABLE will start at the top and proceed by attempting to match the input mnemonic type with the STRUC_TYPE parameter in successive table entries. The RECSIZE parameter refers to the number of bytes in a single record of the type associated with the STRUC_DESCR.

7.1.3 Dynamic Record Descriptor Table: DYNAM_REC_DESCR_TABLE

This table is a running list of all DYNAM_REC_DESCRs. Each time a new record or array of records is declared, a new DYNAM_REC_DESCR is created. The TABLE_INDEX parameter of the DYNAM_REC_DESCR will be set to the next consecutive index in the DYNAM_REC_DESCR_TABLE; the table index portion of the newly created record's (or array of records') longword(s) will also be set to this value.

7.2 Modules And PDL

8 SAVING AND RESTORING ENVIRONMENT

8.1 Design Description Overview

8.2 Modules And PDL

9 JOURNALING

9.1 Design Description Overview

9.2 Modules And PDL

10 IGORE CONDITION HANDLER

10.1 Design Description Overview

10.2 Modules And PDL

11 AFE PREPROCESSOR

11.1 Design Description Overview

11.2 Modules And PDL

12 STRUCTURE DESCRIPTOR PREPROCESSOR

12.1 Design Description Overview

12.2 Modules And PDL